1.Duplicated Code(重复的代码)

臭味行列中首当其冲的就是Duplicated Code。如果你在一个以上的地点看到相同的程序结构，那么当可肯定：设法将它们合而为一，程序会变得更好。

最单纯的Duplicated Code就是［同一个class内的两个方法含有相同表达式(expression)］。这时候你需要做的就是采用Extract Method提炼出重复的代码，然后让这两个地点都调用被提炼出来的那一段代码。

另一种常见情况就是[两个互为兄弟(sibling)的subclasses内含有相同表达式]。要避免这种情况，只需要对两个classes都使用 Extract Method，然后再对被提炼出的代码使用Pull Up Method，将它推入superclass内。如果代码之间只是类似，并非完全相同，那么就得运用Extract Method将相似部分和差异部分割开，构成单独一个方法。然后你可能发现或许可以运用Form Template Method获得一个Template Method设计模式。如果有些方法以不同的算法做相同的事，你可以择定其中较清晰的一个，并使用Substitute Algorithm将其它方法的算法替换掉。

如果两个毫不相关的classes内出现Duplicated Code，你应该考虑对其中一个使用Extract Class，将重复代码提炼到一个独立class中，然后在另一个class内使用这个新class。但是，重复代码所在的方法也可能的确只应该属于某个 class，另一个class只能调用它，抑或这个方法可能属于第三个class，而另两个classes应该引用这第三个class。你必须决定这个方法放在哪儿最合适，并确保它被安置后就不会再在其它任何地方出现。

2.Long Method(过长方法)

拥有[短方法](short methods)的对象会活得比较好、比较长。不熟悉面向对象技术的人，常常觉得对象程序中只有无穷无尽的delegation(委托)，根本没有进行任何计算。和此类程序共同生活数年之后，你才会知道，这些小小方法有多大价值。［间接层］所能带来的全部利益——解释能力、共享能力、选择能力——都是由小型方法支持的。

很久以前程序员就已认识到：程序愈长愈难理解。早期的编程语言中，［子程序调用动作］需要额外开销，这使得做你们不太乐意使用small method，现代OO语言几乎已经完全免除了进程内的[方法调用动作额外开销]。不过代码阅读者还是得多费力气，因为他必须经常转换上下文去看看子程序做了什么。某些开发环境允许用户同时看到两个方法，这可以帮助你省去部分麻烦，但是让small method容易理解的真正关键在于一个好名字。如果你能给方法起个好名字，读者就可以通过名字了解方法的作用，根本不必去看其中写了些什么。

最终的效果是：你应该更积极进取地分解方法。我们遵循这样一条原则：每当感觉需要以注释来说明点什么的时候，我们就把需要说明的东西写进一个独立的方法中，并以其用途(而非实现手法)命名。我们可以对一组或甚至短短一行代码做这件事。哪怕替换后的方法调用动作比方法自身还长，只要方法名称能够解释其用途，我们也该毫不犹豫地那么做。关键不在于方法的长度，而在于方法[做什么]和[如何做]之间的语义距离。

百分之九十九的场合里，要把方法变小，只需使用Extract Method。找到方法中适合集在一起的部分，将它们提炼出来形成一个新方法。

如果方法内有大量的参数和临时变量，它们会对你的方法提炼形成阻碍。如果你尝试运用Extract Method，最终就会把许多这些参数和临时变量当作参数，传递给被提炼出来的新方法，导致可读性

几乎没有任何提升。啊是的，你可以经常运用 Replace Temp with Query则可以将过长的参数列变得更简洁一些。

如果你已经这么做，仍然有太多临时变量和参数，那就应该拿出我们的杀手锏：Replace Method with Method Object。

如何确定该提炼哪一段代码呢？一个很好的技巧是：寻找注释。它们通常是指出[代码用途和实现手法间的语义距离]的信号。如果代码前言有一行注释，就是在提醒你：可以将这段代码替换成一个方法，而且可以在注释的基础上给这个方法命名。就算只有一行代码，如果它需要以注释来说明，那也值得将它提炼到独立的方法去。

条件式和循环常常也是提炼的信号。你可以使用Decompose Conditional处理条件式。至于循环，你应该将循环和其内的代码提炼到一例独立方法中。

3.Large Class(过大类)

如果想利用单一class做太多事情，其内往往就会出现太多instance变量。一旦如此，Duplicated Code也就接踵而至了。

你可以运用Extract Class将数个变量一直提炼到新class内。提炼时应该选择class内彼此相关的变量，将它们放在一直。例如”depositAmount” 和”depositCurrency”可能应该隶属同一个class。通常如果class内的数个变量有着相同的前缀或字尾，这就意味有机会把它们提炼到某个组件内。如果这个组件适合作为一个subclass，你会发现Extract Subclass往往比较简单。

有时候class并非在所有时刻都使用所有instance变量。果真如此，你或许可以多次使用Extract Class或Extract Subclass。

和[太多instance变量]一样，class内如果有太多代码，也是[]代码重复、混乱、死亡]的绝佳滋生地点。最简单的解决方案是把赘余的东西消弭于class内部。如果有五个[百行方法]，它们之中很多代码都相同，那么或许你可以把它们变成五个［十行方法］和十个提炼出来的［双行方法］。

和[拥有太多instance变量]一样，一个class如果拥有太多代码，往往也适合使用Extract Class和Extract Subclass。这里有个有用技巧：先确定客户端如何使用它们，然后运用Extract Interface为每一种使用一个接口。这或许可以帮助你看清楚如何分解这个class。

如果你的Large Class是个GUI class，你可能需要把数据和行为移到一个独立的领域对象去。你可能需要两边各保留一些重复数据，并令这些数据同步。Duplicate Observed Data告诉你该怎么做。这种情况下，特别是如果你使用旧式AWT组件，你可以采用这种方式去掉GUI class并代以Swing组件。

4.Long Parameter List(过长参数列)

刚开始学习编程的时候，老师教我们：把方法所需的所有东西都以参数传递进去。这可以理解，因为除此之外就只能选择全局数据，而全局数据是邪恶的东西。对象技术改变了这一情况，因为如果你手上没有你所需要的东西，总可以叫另一个对象给你。因此，有了对象，你就不必把方法需要的所有东西都以参数传递给它了，你只需给它足够的东西、让方法能从中获得自己需要的所有东西就

行了。方法需要的东西多半可以在方法的宿主类(host class)中找到。面向对象程序中的方法，其参数列通常比在传统程序中短得多。

这是好现象，因为太长的参数列难以理解，太多参数会造成前后不一致、不易使用，而且一旦你需要更多数据，就不得不修改它。如果将对象传递给方法，大多数修改都将没有必要，因为你很可能只需(在方法内)增加一两条请求，就能得到更多数据。

如果[向既有对象发出一条请求]就可以取得原本位于参数列上的一份数据，那么你应该激活重构准则Replace Parameter with Method。上述的既有对象可能是方法所属class内的一个字段，也可能是另一个参数。你还可以运用Preserve Whole Object将来自同一对象的一堆数据收集起来，并以该对象替换它们。如果某些数据缺乏合理的对象归属，可使用Introduce Parameter Object为它们制造出一个[参数对象]。

此间存在一个重要的例外。有时候你明显不希望造成[被调用之对象]与[较大对象]间的某种依存关系。这时候将数据从对象中拆解出来单独作为参数，也很合情合理。但是请注意其所引发的代价。如果参数列太长或变化太频繁，你就需要重新考虑自己的依存结构了。

5.Divergent Change(发散式变化)

我们希望软件能够更容易被修改——毕竟软件再怎么说本来就该是[软]的。一旦需要修改，我们希望能够跌到系统的某一点，只在该处做修改。如果不能做到这点，你就嗅出两种紧密相关的刺鼻味道中的一种了。

如果某个class经常因为不同的原因在不同的方向上发生变化，Divergent Change就出现了。当你看着一个class说：“ 呃，如果新加入一个数据库，我必须修改这三个方法；如果新出现一种金融工具，我必须修改这四个方法”，那么此时也许将这个对象分成两个会更好，这么一来每个对象就可以只因一种变化而需要修改。当然，往往只有在加入新数据库或新金融工具后，你才能发现这一点。针对某一外界变化的所有相应修改，都只应该发生在单一class中，而这个新class内的所有内容都应该反应该外界变化。为此，你应该找出因着某特定原因而造成的所有变化，然后运用Extract Class将它们提炼到另一个class中。

6.Shotgun Surgery(霰弹式修改)

Shotgun Surgery类似Divergent Change，但恰恰相反。如果每遇到某种变化，你都必须在许多不同的class内做出许多小修改以响应之，你所面临的坏味道就是Shotgun Surgery。如果需要修改的代码散布四处，你不但很难找到它们，也很容易忘记某个重要的修改。

这种情况下你应该使用Move Method和Move Field把所有需要修改的代码放进同一个class。如果眼下没有合适的class可以安置这些代码，就创造一个。通常你可以运用Inline Class把一系列相关行为放进同一个class。这可能会造成少量Divergent Change，但你可以轻易处理它。

Divergent Change是指[一个class受多种变化的影响]，Shotgun Surgery则是指[一种变化引发多个classes相应修改]。这两种情况下你都会希望整理代码，取得[外界变化]与[待改类]呈现一对一关系的理想境地。

7.Feature Envy(依恋情结)

对象技术的全部要点在于：这是一种[将数据和加诸其上的操作行为包装在一起]的技术。有一种经典气味是：方法对某个class的兴趣高过对自己所处之 host class的兴趣。这种孺慕之情最通常的焦点便是数据。无数次经验里，我们看到某个方法为了计算某值，从另一个对象那儿调用几乎半打的取值方法。疗法显而易见：把这个方法移到另一个地点。你应该使用Move Method把它移到它该去的地方。有时候方法中只有一部分受这种依恋之苦，这时候你应该使用Extract Method把这一部分提炼到独立方法中，再使用Move Method带它去它的梦中家园。

当然，并非所有情况都这么简单。一个方法往往会用上数个classes特性，那么它究竟该被置于何处呢？我们的原则是：判断哪个class拥有最多[被此方法使用]的数据，然后就把这个方法和那些数据摆在一起。如果先以Extract Method将这个方法分解为整个较小方法并分别置放于不同地点，上述步骤也就比较容易完成了。

有数个复杂精巧的模式破坏了这个规则。说起这个话题，［四巨头］的Streategy和Visitor立刻跳入我的脑海，Kent Beck的Self Delegation也丰此列。使用这些模式是为了对抗坏味道Divergent Change。最根本的原则是：将总是一起变化的东西放在一块儿。［数据］和[引用这些数据]的行为总是一起变化的，但也有例外。如果例外出现，我们就搬移那些行为，保持[变化只在一起发生]。Strategy和Visitor使你得以轻松修改方法行为，因为它们将少量需要被覆写的行为隔离开来——当然也付出了[多一层间接性]的代价。

8.Data Clumps(数据泥团)

数据项就像小孩子：喜欢成群结队地待在一块儿。你常常可以在很多地方看到相同的三或四笔数据项：两个classes内的相同字段、许多方法签名式中的相同参数。这些[总是绑在一起出现的数据]真应该放进属于它们自己的对象中。首先请找出这些数据的字段形式出现点，运用Extract Class将它们提炼到一个独立对象中。然后将注意力转移到方法签名式上头，运用Introduce Parameter Object或Preserve Whole Object为它减肥。这么做的直接好处是可以将很多参数列缩短，简化方法调用动作。是的，不必因为Data Clumps只用上新对象的一部分字段而在意，只要你以新对象取代两个(或更多)字段，你就值回票价了。

一个好的评断办法是：删掉众多数据中的一笔。其它数据有没有因而失去意义？如果它们不再有问询，这就是个明确信号：你应该为它们产生一个新对象。

缩短字段个数和参数个数，当然可以支队一些坏味道，但更重要的是：一旦拥有新对象，你就有机会让程序散发出一种芳香。得到新对象后，你就可以着手寻找 Feature Envy，这可以帮你指出[可移到新class]中的种种程序行为。不必太久，所有classes都将在它们的小小社会中充分发挥自己的生产力。

9.Primitive Obsession(基本型别偏执)

大多数编程环境都有两种数据：结构型别允许你将数据组织成有意义的形式；基本型别则是构成结构型别的积木块。结构总是会带来一定的额外开销。它们有点像数据库中的表格，或是那些得不偿失的东西。

对象的一个极具价值的东西早到：它们模糊了横亘于基本数据和体积较大的classes之间的界限。你可以轻松编写出一些与语言内置型别无异的小型 classes。例如Java就以基本型别表示数值，而心class表示字符串和日期——这两个型别在其它许多编程环境中都以基本型别表现。

对象技术的新手通常在小任务上运用小对象——像是结合数值和币别的money class、含一个起始值和一个结束值的range class、电话号码或邮政编码等等的特殊strings。你可以运用Replace Data Value with Object将原本单独存在的数据值替换为对象，从而走出传统的洞窟，进入炙手可热的对象世界。如果欲替换之数据值是type code，而它并不影响行为，你可以运用Replace Type Code with Class将它换掉。如果你有相依于此type code的条件式，可运用Replace Type Code with Subclass或Replace Type Code with State/Strategy加以处理。

如果你有一组应该总是被放在一起的字段，可运用Extract Class。如果你在参数列中看到基本型数据，不妨试试Introduce Parameter Object。如果你发现自己正从array中挑选数据，可运用Replace Array with Object。

10.Switch Statements(switch惊悚现身)

面向对象程序的一个最明显特征就是：少用switch(或case)语句。从本质上说，switch语句的问题在于重复。你常会发现同样的switch语句散布于不同的地点。如果要为它添加一个新的case子句，你必须找到所有switch语句并修改它们。面向的多态概念可为此带来优雅的解决办法。

大多数时候，一看到switch语句你就应该考虑以多态来替换它。问题是多态该出现在哪儿？switch语句常常根据type code进行选择，你要的是[与该type code相关的方法或class]。所以你应该使用Extract Method将switch语句提炼到一个独立方法中，再以Move Method将它搬移到需要多态性的那个class里头。此时你必须决定是否使用Replace Type Code with Subclasses或Replace Type Code with State/Strategy。一旦这样完成继承结构之后，你就可以运用Replace Conditional with Polymorphism了。

如果你只是在单一方法中髭选择事例，而你并不想改动它们，那么[多态]就有点杀鸡用牛刀了。这种情况下Replace Parameter with Explicit Methods是个不错的选择。如果你的选择条件之一是null，可以试试Introduce Null Object。

11.Parallel Inheritance Hierarchies(平等继承体系)

Parallel Inheritance Hierarchies其实是Shotgun Surgery的特殊情况。在这种情况下，每当你为某个class增加一个subclass，必须也为另一个class相应增加一个subclass。如果你发现某个继承体系的class名称前缀和另一个继承体系的class名称前缀完全相同，便是闻到了这种坏味道。

消除这种重复性的一般策略是：让一个继承体系的实体指涉另一个继承体系的实体。如果再接再厉运用Move Method和Move Field，就可以将指涉端的继承体系消弭于无形。

12.Lazy Class(冗赘类)

你所创建的每一个class，都得有人去理解它、维护它，这些工作都是要花钱的。如果一个class的所得不值其身份，它就应该消失。项目中经常会出现这样的情况：某个class原本对得起自己的身份，

但重檐使它身形缩水，不再做那么多工作；或开发者事前规划了某些变化，并添加一个class来就会这些变化，但变化实际上没有发生。不论上述哪一种原因，请让这个class庄严赴义吧。如果某些subclass没有做满足够工作，试试Collapse Hierarchy[合并继承]。对于几乎没用的组件，你应该以Inline Class对付它们。

13.Speculative Generality(夸夸其谈未来性)

这个令我们十分敏感的坏味道，命名者是Brian Foote。当有人说“噢，我想我们总有一天需要做这事”并因而企图以各式各样的挂勾和特殊情况来处理一些非必要的事情，这种坏味道就出现了。那么做的结果往往造成系统更难理解和维护。如果所有装置都会被用到，那就值得那么做；如果用不到，就不值得。用不上的装置只会挡你的路，所以，把它搬弄吧。

如果你的某个abstract class其实没有太大作用，请运用Collapse Hierarchy。非必要之delegation可运用Inline Class除掉。如果方法的某些参数示被用上，可对它实施Rename Method让它现实一些。

如果方法或class的惟一用户是test cases，这就飘出了坏味道Speculative Generality。如果你发现这样的方法或class，请把它们连同其test cases都删掉。但如果它们的用途是帮助test cases检测正当功能，当然必须刀下留人。

14.Temporary Field(令人迷惑的暂时字段)

有时你会看到这样的对象：其内某个instance 变量仅为某种特定情势而设。这样的代码让人不易理解，因为你通常认为对象在所有时候都需要它的所有变量。在变量未被使用的情况下猜测当初其设置目的，会让你发疯。

请使用Extract Class给这个可怜的孤独创造一个家，然后把所有和这个变量相关的代码都放进这个新家。也许你还可以使用Introduce Null Object在[变量不合法]的情况下创建一个Null对象，从而避免写出[条件式代码]。

如果class中有一个复杂算法，需要好几个变量，往往就可能导致坏味道Temporary Field的出现。由于实现者不希望传递一长串参数，所以他把这些参数都放进字段中。但是这些字段只在使用该算法时才有效，其它情况下只会让人迷惑。这时候你可以利用Extract Class把这些变量和其相关方法提炼到一个独立class中。提炼后的新对象将是一个method object。

15.Message Chains(过度耦合的消息链)

如果你看到用户向一个对象索求另一个对象，然后再向后者索求另一个对象，然后再索求另一个对象……这就是Message Chain。实际代码中你看到的可能是一长串getThis()或一长串临时变量。采取这种方式，意味客户将与查找过程中的航行结构紧密耦合。一旦对象间的关系发生任何变化，客户端就不得不做出相应修改。

这时候你应该使用Hide Delegate。你可以在Message Chain的不同位置进行这种重构手法。理论上你可以重构Message Chain上的任何一个对象，但这么做往往会把所有中介对象都变成Middle Man。通常更好的选择是：先观察Message Chain最终得到的对象是用来干什么的，看看能否以Extract Method把使用该对象的代码提炼到一个独立方法中，再运用Move Method把这个方法推入

Message Chain。如果这条链上的某个对象有多位客户打算航行此航线的剩余部分，就加一个方法来做这件事。

有些人把任何方法链都视为坏东西，我们不这样想。呵呵，我们的总代表镇定是出了名的，起码在这件事情上是这样。

16.Middle Man(中间转手人)

对象的基本特征之一就是封装——对外部世界隐藏其内部细节。封装往往伴随delegation。比如说你问主管是否有时间参加一个会议，他就把这个消息委托给他的记事簿，然后才能回答你。很好，你没必要知道这位主管到底使用传统记事簿或电子记事簿抑或秘书来记录自己的约会。

但是人们可能过度运用delegation。你也许会看到某个class接口有一半的方法都委托给其它class，这样就是过度运用。这里你应该使用 Remove Middle Man，直接和负责对象打交道。如果这样[不干实事]的方法只有少数几个，可以运用Inline Method把它们"inlining"，放进调用端。如果这些Middle Man还有其它行为内销可以运用Replace Delegation with Inheritance把它变成负责对象的subclass，这样你既可以扩展原对象的行为，又不必负担那么多的委托动作。

17.Inappropriate Intimacy(狎昵关系)

有时候你会看到两个classes过于亲密，花费太多时间去探究彼此的private成分。如果这发生在两个[人]之间，我们不必做卫道之士；但对于 classes，我们希望它们严守清规。

就像古代恋人一样，过份狎昵的classes必须拆散。你可以采用Move Method和Move Field帮它们划清界线，从而减少狎昵行径。你也可以看看是否运用Change Bidirectional Association to Unidirectional[将双向关联改为单向]让其中一个class对另一个斩断情丝。如果两个classes实在情投意合，可以运用Extract Class把两者共同点提炼到一个安全地点，让它们坦荡地使用这个新class。或者也可以尝试运用Hide Delegate让另一个class来为它们传递相思情。

继承往往造成过度亲密，因为subclass对superclass的了解总是超过superclass的主观愿望。如果你觉得该让这个孩子独自生活了，请运用Replace Inheritance with Delegation让它离开继承体系。

18.Alternative Classes with Different Interfaces(异曲同工的类)

如果两个方法做同一件事，却有着不同的签名式，请运用Rename Method根据它们的用途重新命名。但这往往不够，请反复运用Move Method将某些行为移入classes，直到两者的协议一致为止。如果你必须重复而赘余地移入代码才能完成这些，或许可运用Extract Superclass为自己赎点罪。

19.Incomplete Library Class(不完美的程序库类)

复用常被视为对象的终极目的。我们认为这实在是过度估计了。但是无可否认，许多编程技术都建立在library classes的基础上，没人敢说是不是我们都把排序算法忘得一干二净了。

Library classes构筑者没有未卜先知的能力，我们不能因此责怪他们。毕竟我们自己也几乎总是在系统快要构筑完成的时候才能弄清楚它的设计，所以 library构筑者的任务真的很艰巨。麻烦的是library的形式往往不够好，往往不可能让我们修改其中的classes使它完成我们希望完成的工作。这是否意味那些经过实践检验的战术如Move Method等等，如今都派不上用场了？

幸好我们有两个专门就会这种情况的工具。如果你只想修改library classes内的一两个方法，可以运用Introduce Foreign Method；如果想要添加一大堆额外行为，就得运用Introduce Local Extension。

20.Data Class(纯稚的数据类)

所谓Data Class是指：它们拥有一些字段，以及用于访问这些字段的方法，除此之外一无长物。这样的classes只是一种[不会说话的数据容器]，它们几乎一定被其它classes过份细琐地操控着。这些classes早期可能拥有public字段，果真如此你应该在别人注意到它们之前，立刻运用 Encapsulate Field将它们封装起来。如果这些classes内含容器类的字段，你应该检查它们是不是得到了恰当的封装；如果没有，就运用Encapsulate Collection把它们封装起来。对于那些不该被其它classes修改的字段，请运用Remove Setting Method。

然后，找出这些[取值/设值]方法被其它classes运用的地点。尝试以Move Method把那些调用行为搬移到Data Class来。如果无法搬移整个方法，就运用Extract Method产生一个可被搬移的方法。不久之后你就可以运用Hide Method把这些[取值/设值]方法隐藏起来了。

Data Class就像小孩子。作为一个起点很好，但若要让它们像[成年]的对象那样参与整个系统的工作，它们就必须承担一定责任。

21.Refused Bequest(被拒绝的遗赠)

Subclasses应该继承superclass的方法和数据。但如果它们不想或不需要继承，又该怎么办呢？它们得到所有礼物，却只从中挑选几样来玩！

按传统说法，这就意味继承体系设计错误。你需要为这个subclass新建一个兄弟，再运用Push Down Method和Push Down Field把所有用不到的方法下推给那兄弟。这样一来superclass就只持有所有subclasses共享的东西。常常你会听到这样的建议：所有 superclasses都应该是抽象的。

既然使用[传统说法]这个略带贬义的词，你就可以猜到，我们不建议你这么做，起码不建议你每次都这么做。我们经常利用subclassing手法来复用一些行为，并发现这可以很好地应用于日常工作。这也是一种坏味道，我们不否认，但气味通常并不强烈。所以我们说：如果Refused Bequest引起困惑和问题，请遵循传统忠告。但不必认为你每次都得那么做。十有八九这种坏味道很淡，不值得理睬。

如果subclass复用了superclass的行为(实现)，却又不愿意支持superclass的接口，Refused Bequest的坏味道就会变得浓烈。拒绝继承superclass的实现，这一点我们不介意；但如果拒绝继承superclass的接口，我们不以为然。不过即使你不愿意继承接口，也不要胡乱修改继承系，你应该运用Replace Inheritance with Delegation来达到目的。

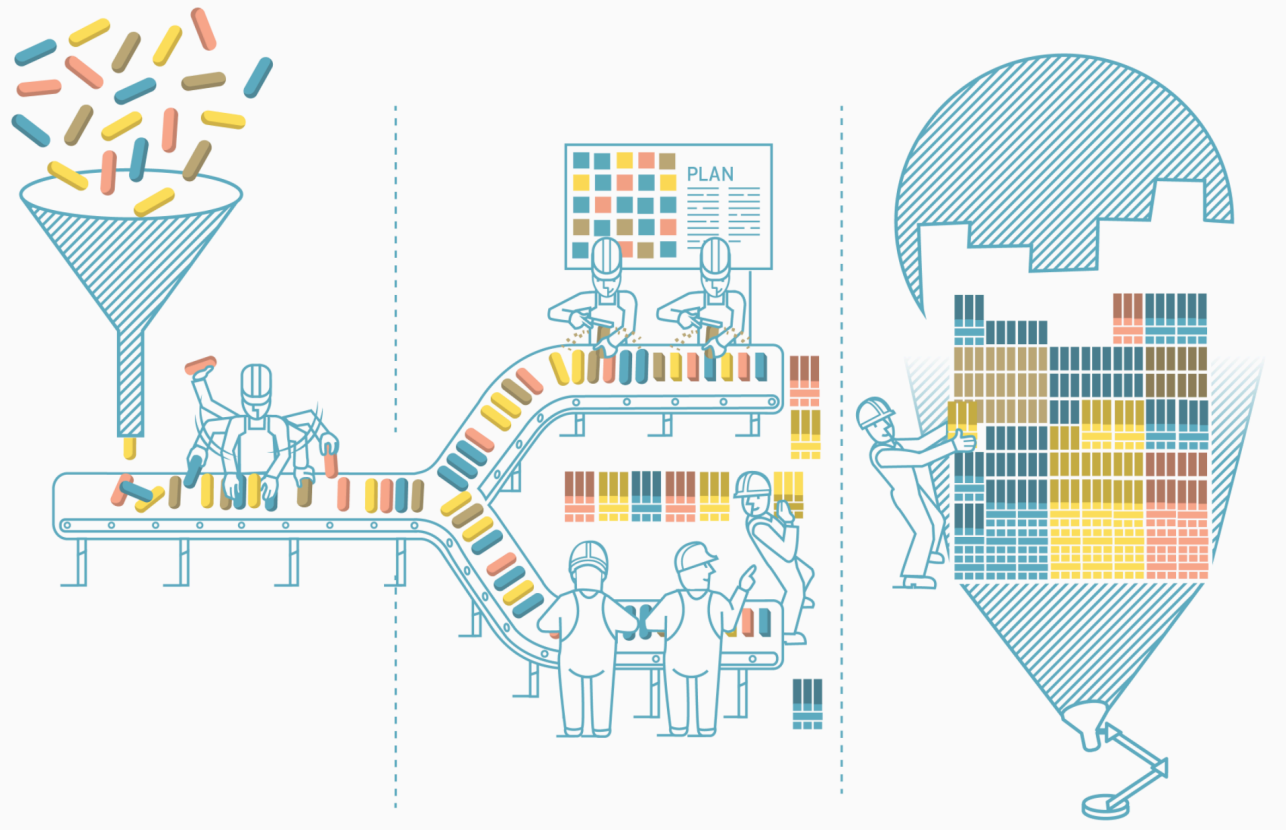
22.Comments(过多的注释)

别担心，我们并不是说你不该写注释。从嗅觉上说，Comments不是一种坏味道；事实上它们还是一种香味呢。我们之所以要在这里提到Comments，因为人们常把它当作除臭剂来使用。常常会有这样的情况：你看到一段代码有着长长的注释，然后发现，这些注释之所以存在乃是因为代码很糟糕。这种情况的发生次数之多，实在令人吃惊。

Comments可以带我们找到本章先前提到的各种坏味道。找到坏味道后，我们首先应该以各种重构手法把坏味道去除。完成之后我们常常会发现：注释已经变得多余了，因为代码已经清楚说明了一切。

如果你需要注释来解释一块代码做了什么，试试Extract Method;如果你需要注释说明某些系统的需求规格，试试Introduce Assertion。

如果你不知道该做什么，这才是注释的良好运用时机。除了用来记述将来的打算之外，注释还可以用来标记你并无十足把握的区域。你可以在注释里写下自己[为什么做某某事]。这类信息可以帮助将来的修改者，尤其是那些健忘的家伙。



22 code bad smell	22种代码坏味
Duplicated Code	重复的代码
Long Method	过长方法
Large Class	过大类
Long Parameter List	过长参数列
Divergent Change	发散式变化
Shotgun Surgery	霰弹式修改
Feature Envy	依恋情结
Data Clumps	数据泥团
Primitive Obsession	基本型别偏执
Switch Statements	switch惊悚现身
Parallel Inheritance Hierarchies	平等继承体系
Lazy Class	冗赘类
Speculative Generality	夸夸其谈未来性
Temporary Field	令人迷惑的暂时字段
Message Chains	过度耦合的消息链
Middle Man	中间转手人
Inappropriate Intimacy	狎昵关系
Alternative Classes with Different Interfaces	**异曲同工的类**
Incomplete Library Class	不完美的程序库类
Data Class	纯稚的数据类
Refused Bequest	被拒绝的遗赠
Comments	过多的注释

Duplicated Code --------（重复代码）难维护。
[解决方法]：提取公共函数。
Long Method ----------（函数长）难理解。
[解决方法]：拆分成若干函数。
Large Class ----------（类大）难理解。
[解决方法]：拆分成若干类。
Long Parameter List ----（参数多）难用，难理解。

[解决方法]：将参数封装成结构或者类。
Divergent Change -------（万能类）发散试修改,改好多需求，都会动他。
[解决方法]：拆，将总是一起变化的东西放在一块儿。
Shotgun Surgery --------（天女散花的逻辑）散弹试修改,改某个需求的时候，要改很多类。
[解决方法]：将各个修改点，集中起来，抽象成一个新类。
Feature Envy -----------（红杏出墙的函数）使用了大量其他类的成员
[解决方法]：将这个函数挪到那个类里面。
Data Clumps ------------（数据团）常一起出现的一坨数据。
[解决方法]:他们那么有基情，就在一起吧，给他们一个新的类。
Primitive Obsession ----（偏爱基本类型）热衷于使用int,long,String等基本类型。
[解决方法]:反复出现的一组参数，有关联的多个数组换成类吧。
Switch Statements ------（switch语句）
[解决方法]:state/strategy 或者只是简单的多态。
Parallel Inheritance Hierarchies ----------（平行继承）增加A类的子类ax，B类也需要相应的增加一个bx。
[解决方法]:应该有一个类是可以去掉继承关系的。
Lazy Class ------------------------------------（冗赘类）如果他不干活了，炒掉他吧。
[解决方法]:把这些不再重要的类里面的逻辑，合并到相关类，删掉旧的。
Speculative Generality -------------------------（夸夸其谈未来性）
[解决方法]:删掉
Temporary Field --------------------------------（临时字段）仅在特定环境下使用的变量
[解决方法]:将这些临时变量集中到一个新类中管理。
Message Chains ---------------------------------（消息链）过度耦合的才是坏的。
[解决方法]:拆函数或者移动函数。
Middle Man -------------------------------------（中介）大部分都交给中介来处理了。
[解决方法]:用继承替代委托。
Inappropriate Intimacy -------------------------（太亲密）两个类彼此使用对方的私有的东西。
[解决方法]:划清界限拆散，或合并，或改成单项联系。
Alternative Classes with Different Interfaces --（相似的类，有不同接口）
[解决方法]:重命名，移动函数，或抽象子类。
Incomplete Library Class ----------------------（不完善的类库）
[解决方法]:包一层函数或包成新的类。
Data Class -------------（纯数据类）类很简单，仅有公共成员变量，或简单操作函数。
[解决方法]:将相关操作封装进去，减少public成员变量。
Refused Bequest --------（继承过多）父类里面方法很多，子类只用有限几个。
[解决方法]:用代理替代继承关系。
Comments ----------------（太多注释）这里指代码太难懂了，不得不用注释解释。
[解决方法]:避免用注释解释代码，而是说明代码的目的，背景等。好代码会说话。